Security Checkers: Detecting Processor Malicious Inclusions at Runtime

Michael Bilzor, Ted Huffmire, Cynthia Irvine, and Tim Levin
U.S. Naval Postgraduate School

***Abstract*—To counter the growing threat of malicious subversions to the design of a microprocessor, there is a great need for simple, automated methods for detecting such malevolent changes. Based on the adoption of the Property Specification Language (PSL) for behavioral verification, and the advent of tools for automatically generating synthesizable hardware design language (HDL) constructs for verifying a PSL assertion, we propose a new method called Security Checkers, which uses security-focused PSL assertions to create hardware design units for detecting malicious inclusions at runtime.**

We describe the process flow for creating Security Checkers and demonstrate by example how they can be used to detect malicious inclusions in a processor design. Because the checkers can be used in simulation, FPGA emulation, or as part of a fabricated design, we illustrate how this technique can be used to detect malicious inclusions over a much broader segment of the processor development lifecycle, compared to existing methods.

I. Introduction

General purpose processors are used widely in high-assurance applications, such as in classified networks, sophisticated military hardware, and critical infrastructure. Though most general-purpose processors are designed in the U.S., some even in certified trusted design facilities, vulnerabilities to microprocessor design exist at many stages of the acquisition chain.

In general, introducing a malicious inclusion (MI)[1] into a real processor is difficult, tending to require the resources of a large, well-supported organization, whereas malicious software can be generated easily. However, a physical subversion may be a serious threat, since it can bypass all software defenses and give an attacker complete control over the target system.

A. Description of the Threat

A typical general-purpose processor may be designed in the U.S., fabricated in Taiwan or China, shipped to another country for assembly onto a printed circuit board, then installed in a finished product at yet another location. Malicious changes to a processor can be introduced during any of these phases.

B. Fake Chips in the Supply Chain

Once subverted, a processor could easily find its way into a high-assurance application, given the difficulty of securing the enormous and complex processor supply chain. Most often, lower-quality fake processors with comparable functionality are re-labeled and sold in place of higher-priced originals. Quoting a recent newspaper report: [21]

- From November 2007 through May 2010, U.S. Customs officials said they seized 5.6 million counterfeit chips.
- Two men indicted in October, 2010 admitted importing from China more than 13,000 fake chips altered to resemble those from legitimate companies, including Intel, Atmel, Altera and National Semiconductor. Among those buying the chips was the U.S. Navy.

The motivation for these fakes is apparently economic, and the replacement processors usually perform the intended function (albeit with poorer performance), but they underscore the possibility that maliciously modified processors could be surreptitiously introduced into the supply chain and end up in targeted high-assurance applications.

C. Examples of Malicious Inclusions

Though real-world occurrences of MIs may be kept from the public, some reports do occasionally surface. For example, according to a New York Times article [23], during a 2008 Israeli raid on a suspected Syrian nuclear facility, the Syrian air defenses may have been disabled by a “kill switch” that had been surreptitiously introduced.

More recently, researchers disclosed the existence of a hidden hardware-coded password that provides access to undocumented machine status registers in some AMD processors [6], [5], though it’s not yet clear what level of control can be gained by accessing these registers.

Researchers have explored various avenues for creating processor MIs, as well. Jin, Kupp, and Makris showed several methods for extracting a secret key from an encryption processor [20]. King et al. designed modifications that allow an adversary to conduct “escalation of privilege” and “shadow mode” attacks on a processor [22].

II. Previous Work

Existing methods for detecting malicious changes to a processor’s design generally fall into two categories: physical analysis and design analysis.

A. Physical Analysis

Physical analysis of a processor is either destructive or nondestructive. Wang, Tehranipoor, and Plusquellic described the various physical analysis methods [28], summarized below.

[1] Often called “Hardware Trojans,” we refer to subversions in hardware instead as “malicious inclusions,” in order to preserve a semantic distinction from the Trojan Horse, which requires some action of naive acceptance by the object of the attack, like opening a malicious attachment or hyperlink.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **2011**	2. REPORT TYPE	3. DATES COVERED **00-00-2011 to 00-00-2011**

4. TITLE AND SUBTITLE **Security Checkers: Detecting Processor Malicious Inclusions at Runtime**	5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Department of Computer Science,Monterey,CA,93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S) 11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

in Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on, 2011, pp. 34-39

14. ABSTRACT

To counter the growing threat of malicious subversions to the design of a microprocessor, there is a great need for simple, automated methods for detecting such malevolent changes. Based on the adoption of the Property Specification Language (PSL) for behavioral verification, and the advent of tools for automatically generating synthesizable hardware design language (HDL) constructs for verifying a PSL assertion, we propose a new method called Security Checkers, which uses security-focused PSL assertions to create hardware design units for detecting malicious inclusions at runtime. We describe the process flow for creating Security Checkers and demonstrate by example how they can be used to detect malicious inclusions in a processor design. Because the checkers can be used in simulation, FPGA emulation, or as part of a fabricated design, we illustrate how this technique can be used to detect malicious inclusions over a much broader segment of the processor development lifecycle, compared to existing methods.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**6**	

1) Destructive Techniques: To facilitate physical analysis, a processor must undergo backside thinning. Once the active layers are exposed, sophisticated imaging equipment is used to map the processor's physical layout, which can then be compared against a reference specification.

The obvious limitation of invasive imaging methods is the impracticality of using it on any more than a small representative sample of chips. Also, the resolution of even the most expensive and advanced imaging technology lags slightly behind state-of-the-art processor feature sizes [8].

2) Nondestructive Techniques: Some malicious inclusions can be detected without destroying the processor under inspection. These methods involve stimulating the processor's inputs with digital or analog test patterns, and observing the outputs. The electrical and timing output characteristics form a kind of "fingerprint" for that processor, which can be compared against the fingerprint of a "golden" reference chip for the presence of observable deviations [19], [25].

These methods can be useful for detecting larger malicious inclusions, but may not detect changes affecting less than around .01%[2] of a processor's circuitry [11], and they also rely on the existence of the "golden" reference chip for comparison.

B. Design Analysis

1) Applying Information Flow Controls: Algorithms have been proposed [27] for constraining a hardware design in order to prevent information flow between entities, when one device must process data at multiple levels of confidentiality or integrity. Though not specifically engineered around malicious inclusions, information-flow control techniques can be used to eliminate channels between multi-level data or processes, regardless of whether the channels were maliciously introduced or were inadvertent design artifacts. However, gate-level information flow engineering does not eliminate MIs that employ non-information-flow attacks, such as the previously mentioned examples from King et al. [22].

2) Detecting Design Subversions: Because processor designs are large and complex, are the product of many contributing engineers, and often use third-party intellectual property, malicious inclusions can be introduced into a high-level design while it is being developed. The Defense Advanced Research Projects Agency is sponsoring research in improving our ability to deduce the trustworthiness of a processor design [1].

One method for divining the existence of MIs in an HDL design was proposed by Hicks, et al. [17]. During the design verification phase, they identify sections of the design code that are not triggered by the verification tests. They bypass the "suspicious" circuits with a trigger to an exception-handling routine, using software emulation instead. The method, called Blue Chip, can detect malicious changes to a design and obviates them with a small overhead; however, it requires dedicated support from the operating system (it is a combined hardware-software approach), relies on a test suite to flag the suspicious circuits, and does not detect malicious modifications made after the HDL design phase, such as changes to the synthesized netlists.

In the next section, we propose a new design analysis method, called Security Checkers, for detecting the presence of MIs in a processor design, across a broader cross-section of a processor's lifecycle.

III. Security Checker Methodology

A. Introduction

Security Checkers are an application of recently developed techniques for automatically generating synthesizable hardware entities that verify temporal logic properties. Specifically, a property in a temporal logic language like the Property Specification Language can be verified at runtime using HDL constructs that are automatically generated by specially designed tools, as described later.

The principal idea behind the Security Checker method is to adapt these recently-developed verification tools to facilitate the creation of *security-relevant* runtime checkers which are capable of detecting the action of MIs. Functional verification and MI detection in a processor are related, but complementary - functional verification establishes that the processor correctly executes the specified functionality (arithmetic, jumps, I/O interrupts, etc.), whereas MI detection looks for two things:

- The presence of additional functionality that is not specified in the architectural specification, and
- The modification of existing functionality that may cause it to deviate from that permitted by the security-relevant features of the architectural specification, such as privilege levels, memory segment controls, or special registers.

It is worth emphasizing here that the *architectural specification,* not the processor design, is the baseline for what is and is not "malicious" behavior in a processor. In general, an architectural specification will define the components and instruction set and how they are to behave, but leave the details to the designers. As a result, an x86 chip from Intel and an x86 chip from AMD, for example, may be functionally equivalent but potentially quite different "under the hood."

B. Formal Methods, Model Checking, and Runtime Verification

Processor evaluation generally falls into three categories - formal methods, model checking, and runtime verification.

Formal Methods verification involves proving logically specified properties using tools like Isabelle [2]. Not all interesting properties of a system are statically decidable, however, and complex proofs can be tedious.

Model Checking employs several methods for verifying the correctness of a design. For processor verification, SystemVerilog and the Open Verification Methodology (OVM) have gained in popularity [18]. Because they rely to a great extent on random generation of test stimuli, however, such methods are well suited for verification (since they are likely to uncover

[2] Optimization techniques that an adversary can use to keep an attack relatively small, even below this threshold, were illustrated by Jin, Kupp, and Makris [20].

a logical flaw), but *not* well suited for detecting MIs that may be triggered by a rare event. For example, an MI triggered only by a specific 64-bit value appearing in an opcode or on a data bus might require generating 2^{64} test stimuli to be certain it is activated and detected.

In the context of triggering and detecting MIs, *Runtime Verification* may also suffer from the need to trigger a stimulus event that is very rare, but it does have one advantage over the other methods - it can be performed even when a system is in fielded operation, giving it the ability to detect malicious changes made after the high-level design phase. Formal methods and model checking of a high-level design will not detect malicious changes made subsequently, such as to a processor's synthesized netlist, mask files, or physical structure. On the other hand, runtime checkers can be synthesized into a processor design, allowing them to detect subversions added after the high-level design phase, assuming the checker itself is not also subverted.

C. PSL Background

Because we plan to use a portion of PSL to characterize what constitutes malicious behavior, a brief description is in order.

PSL was adopted as an IEEE standard [7] in 2005, based on work done by Accellera and IBM. It is described in the specification as "a standard language for specifying electronic system behavior." PSL is organized into four "layers" - the boolean, temporal, verification, and modeling layers, each building upon the other; the boolean and temporal layers are of primary interest here.

The *boolean* layer allows the construction of propositional logic expressions; the propositional variables model circuits' binary 0/1 values in a design. The *temporal* layer is used to describe complex temporal relations between those signals, evaluated over several clock cycles. The formulas in the temporal layer are called sequential expressions; together with boolean expressions, they are built up into *properties*. Each property is a description of how some portion of the system should behave. Properties may then be *asserted*, or required to hold, during verification.

We focus on PSL's Foundation Language (FL), which derives from linear temporal logic, or LTL[3]. In a linear-time logic, observable events are ordered in time, and sequences of events do not branch. Eisner gives justification [14] for focusing on linear-time logic, to the exclusion of branching logic, in runtime verification:

> The complexity of branching time model checking is better than that of linear time model checking... [but] only linear time makes sense for dynamic runtime verification... The overlap between linear and branching time is a large one, and the vast majority of properties used in practice belong to the overlap.

Fortunately, as noted in the IEEE PSL standard [7], any specification in the FL can be compiled down to an LTL formula[4], which will be useful in Section III-E.

D. Security Assertions

We define a *security assertion* as an asserted PSL FL property that contains one or more security-relevant invariants which map from the architectural specification to the processor design. The process is outlined in the following steps:

- Identify security-relevant features of the specification, such as privilege levels, protected registers, etc.
- From the processor design, identify the circuits which perform these security-relevant functions.
- In terms of the identified circuits, create PSL assertions that define security compliance, as described in the architectural specification.
- Using the available automated tools, translate the security assertions into synthesizable HDL code.
- Attach the created HDL entities (Security Checkers) to the design, and use them to complement functional verification, typically in simulation or FPGA emulation.
- If desired, leave the Security Checkers in the design, and fabricate them along with the processor, to facilitate detection of MIs in real time.

The following are a few examples of security invariants that might be mapped from an architectural specification to a processor design using security assertions:

- Only a process running in privileged mode may modify the control/special registers.
- Execution transfers from user mode to kernel mode must only occur through specifically defined processor gates, interrupt calls, or exceptions.
- A memory segment labeled with a certain privilege level may not be modified or read by a process labeled with a lower privilege level.

Though it appears in these examples that we are duplicating the processor's own protection logic, in fact the Security Checker derives its form primarily from the *specification*, not from the processor design; only the named circuits involved, which are passively observed by the Checker, come from the processor design. How exactly the security requirement is implemented internal to the processor design is transparent to the Checker; it merely verifies that the implementation executes in a manner faithful to the *specification,* as described by the assertion.

E. Synthesis of Runtime Security Checkers

The idea of runtime checkers for acceptance of properties specified in LTL and PSL builds on work by Pnueli, who described the construction of "Testers" [24] for dynamic verification. The idea of using fabricated finite state machines as checkers specifically for processor security was mentioned conceptually by Abramovici and Bradley [10], but they left open the details of the FSM construction.

[3] In PSL, the LTL operators *G, F, W, U, X,* and *X!* are replaced by their descriptive equivalents *always (never), eventually!, until, until!, next,* and *next!*, where ! indicates the strong form of an operator, rather than the weak form.

[4] With the possible addition of some auxiliary HDL code, referred to as a satellite.

1) FL to LTL to Buchi Automata: The ability to translate FL properties into LTL, noted earlier, is important because it allows us to leverage a body of previous work on the efficient translation of LTL to Buchi Automata [16]. As observed by Pnueli, Gastin, and others, the resulting Buchi Automata can be used for runtime checking of whether the original LTL formula holds over a given sequence.

2) FOCs, MBAC, SynPSL: Before PSL was formalized, Abarbanel, et al., developed a system called Formal Checkers (FoCs) for converting some logic formulas into equivalent, simulatable VHDL [9]. Boule and Zilic improved on FoCs in both efficiency and breadth [13]. Another conversion tool, SynPSL, was developed by Findenig for a subset of the FL [15]. Pnueli, Boule, and Findenig all use a form of Buchi Automata (BA) as an intermediate format to translate logic formulas into equivalent synthesizable HDL entities[5] [15], [13], [24]. See Figure 1 for a diagram of the workflow for creating Security Checkers, using one of the aforementioned tools.

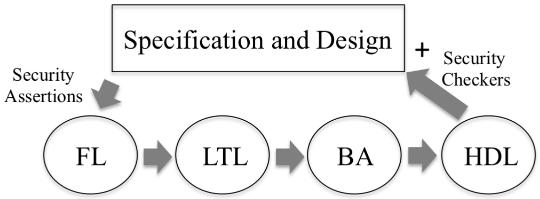

Figure 1. Workflow for the Security Checkers methodology.

IV. Demonstration

A. Processor Design

For this demonstration, we chose the Plasma CPU, by Rhoads, from OpenCores.org [4]. The Plasma is a system-on-chip processor design that implements most of the instructions in the 32-bit MIPS-1 architecture [3]. It has a pipelined execution unit, 32 registers, and a single cache, and provides support for hardware and software interrupts. It has components for serial port communication and Ethernet. We implemented the Plasma on a Spartan-3a FPGA.

B. Example Malicious Inclusion

Since it is a simple processor, designed for experimentation and smaller applications, the Plasma does not implement some of the more sophisticated MIPS functionality, such as *kernel* and *user* modes, and protected virtual memory [3]. Therefore, the MI we designed was relatively subtle - we implemented a *triggered opcode stealer* - when active, it replaces the functionality of an unused, deprecated MIPS opcode. The "stolen" opcode, BEQL, is *branch if equal, likely,* [6] which is not implemented in the normal Plasma design. Until the MI is triggered, the BEQL instruction is ignored, as in a NOP. Once triggered, the BEQL instruction instead mimics the functionality of a *store word* (SW) instruction, allowing the adversary to covertly write data to an arbitrary location in memory. By stealing the opcode in this way, any of the normal memory protection mechanisms implemented via the legitimate memory-access functions are bypassed.

The on and off triggers of our MI would not be found in "normal" machine code; they are intentionally designed as machine instructions no sensible compiler would ever create. The on-trigger is the instruction ORI r0, r0, 0, which takes the contents of register zero (which is hardwired to zero), executes a logical OR with the immediate value 0, and places the result back in r0 (an action which would be ignored anyway). Since it accomplishes no useful work, a compiler would not generate this instruction. Any random 32-bit value could be used as the immediate operand, of course, rather than just 0. As demonstrated by King and Hicks [22], [17], the trigger can be a series of two or more machine-code instructions, making it even more difficult to detect through normal verification methods. Similarly, the off-trigger we use is XORI r0, r0, 0. The BEQL opcode is only "stolen," executing a surreptitious memory operation, if it happens between an on-trigger and an off-trigger.

C. Example Security Checker and Detection

In the Plasma MIPS architecture, the only instructions allowed to write to memory have opcodes between 39 and 44 (the opcode of BEQL is 20). Therefore, the PSL assertion we used to verify that only the proper opcodes can affect the circuits in charge of commanding a write out to memory is:

assert never [((mem_source(0)='1') and not
((opcode>39) and (opcode<44))] @rose(clk)

where *rose* indicates a clock rising edge, and *mem_source(0)='1'* is a condition for propagating a memory write command. This example assertion is a simple boolean expression, but PSL also allows for properties that are sequences of expressions over time, repeated sequences, concatenated sequences, and triggers from one sequence to the another, for example, allowing the construction of rich and complex properties. [7]. Though a processor design may require a significant number of security assertions, especially for a large processor, for brevity we discuss here only the assertion required for detecting MIs of the type in the example. We synthesized the security assertion using SynPSL [15], which automatically constructed a small, complete VHDL entity called checker1.vhd, which we then pasted into the Plasma design. When we ran simulations of arbitrary code execution, the MI remained dormant, and the Security Checker did not detect a fault.

We then ran simulations of the Plasma running executable code that contained the software triggers for the MI. The

[5] Note that modern hardware-design tools are able to accept some PSL specifications and test them in simulation, but these are simulator-specific features, whereas the checkers generated using the tools mentioned above generate synthesizable HDL entities - actual design units that can not only be simulated in any design software, but also emulated in FPGAs and fabricated into silicon.

[6] "Likely" instructions were meant to aid in prediction performance, by denoting an operation such as a conditional branch as being more likely to be taken than not.

malicious logic was activated and performed its backdoor write to memory. The assembly code segment for the demonstration is shown in Figure 2:

```
ORI    $0, $0, 0     ; on-trigger for MI
BEQL   $8, $9, 0     ; surreptitious memory access
XORI   $0, $0, 0     ; off-trigger for MI
BEQL   $8, $9, 0     ; MI is dormant, instr. ignored
```

Figure 2. Assembly Code for MI Trigger and Test. The MI is only active between an on-trigger and an off-trigger.

In the simulation image in Figure 3, the signal checker1_fail goes high at around 9,000ns, at the time the MI is active in the second instruction in Figure 2, but does not go high afterward, since the MI has been disabled by the off-trigger. The failure detection occurs with one clock cycle of latency.

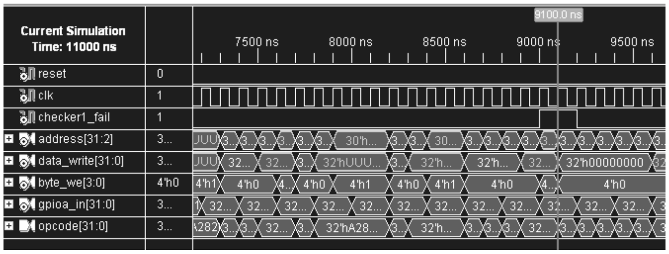

Figure 3. ModelSim demonstration of MI detection. Note checker1_fail, in the third row, goes high at around 9,000ns.

V. Conclusions

Since we designed both the security assertions and the MI in this case, the detection was straightforward. We hope to test a suite of Security Checkers against unknown MIs, produced independently, in other experiments. When synthesized for a Spartan-3a FPGA, the state machine for checker1 had no effect on maximum clock rate, and added approximately .1% of area overhead, in terms of configurable logic blocks, or "slices." We note the following strengths and limitations of the methodology.

Strengths:

- It is the first technique for detecting MIs that is not some form of "equivalence checking" between two designs, since it derives the notion of security compliance directly from a processor's architectural specification.
- Construction of the HDL code for the checkers is automated, thanks to the conversion tools.
- Synthesizable HDL entities can easily be run on all existing simulators.
- The checkers can be added to, and removed from, a target processor's design without affecting its operation.
- It works across simulation, emulation, and fabrication, and detects changes made after the high-level design. The same generated constructs can be used throughout.
- Once the checkers are in place, the target design can be obfuscated without impeding the checkers' operation.
- It can be used to detect MIs which are very small in size, and so not detectable by physical analysis techniques.

Limitations:

- To detect MIs that employ rare-event triggers, there must be some way to discover the trigger. We are interested in integrating HDL "code coverage" tools to assist with this difficult task in simulation checks. On the other hand, if the checkers are left in the processor, rare-event triggered MIs can be captured as they occur, at runtime.
- Some architectural specifications may not contain sufficient details of all permitted and proscribed behaviors.
- Though the conversion tool is automated, the construction of the original PSL security assertions must be done manually, and requires some knowledge of the design.
- The fabricated checkers add power and area overhead. We seek to quantify these requirements in future work.
- Even low-latency detection may not preclude "zero-cycle" attacks, such as an immediate disablement.
- The checkers can be bypassed by an attacker with detailed knowledge of how the monitoring is being performed, if the adversary gains access after the checkers are installed.
- In the general case, properties that are not safety properties (e.g., liveness or fairness properties) may not be Security-Checker enforceable, since a Security Checker is a form of Execution Monitor [26].

A. Comparison to Other Methods

When comparing methods of MI detection across the design lifecycle, there are three important phases to consider. The first is the *baseline reference* stage of the design that is assumed to be secure (not subverted). The second is the design phase during which the detection method is employed, or the *application stage*. The third, the *detection window*, is the portion of the design lifecycle during which malicious modifications (if made in that portion) will be eventually detected - the larger the detection window, the better, for detecting MIs.

For example, physical analysis methods [11], [25] take the "golden sample" chip as a baseline; if both chips come from one design process and the design was corrupted, though, both the reference chip and test chip may match, since both could derive from the same modified design.

The Blue Chip method [17], described earlier, assumes that the design may have been subverted, but trusts that the verification suite will correctly highlight any "suspicious" circuits; the authors note that an incomplete or subverted verification suite would undermine the method. Also, malicious changes made after the high-level design phase (such as changes to the netlist or chip mask) may not be detected.

Another interesting design analysis approach called Trusted RTL was presented by Banga and Hsiao [12]. They compared a trusted behavioral model and an untrusted RTL design, using input test patterns to find suspicious circuits that are rarely triggered, then localizing down to specific differences. Since the approach relies on a SAT-solver, the scalability of an apparently NP-hard detection problem seems an open question. Also, their equivalence-checking method only detects deviations introduced between a high-level behavioral description and an optimized register-transfer level (RTL) description, covering a relatively narrow *detection window* and requiring a trusted high-level design.

The Security Checkers approach, though it also references the high-level design, reaches all the way back to the original architectural specification to help define what is security compliant vs. what is malicious. Since this approach allows the Checkers to be carried forward through simulation, emulation, and fabrication, it can detect MIs over a broad segment of the design lifecycle. The *baseline reference*, *application stage*, and *detection window* for each approach discussed are shown in Figure 4.

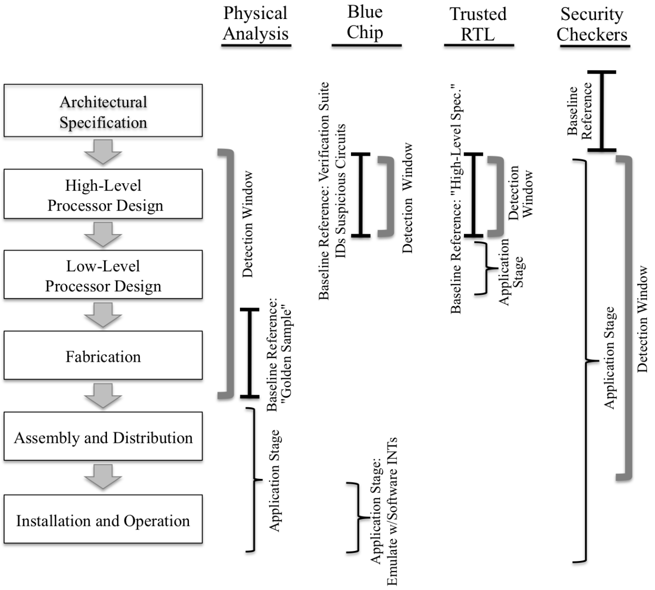
Physical Analysis
Blue Chip
Trusted RTL
Security Checkers
Architectural Specification
High-Level Processor Design
Low-Level Processor Design
Fabrication
Assembly and Distribution
Installation and Operation
Detection Window
Baseline Reference: "Golden Sample"
Application Stage
Baseline Reference: Verification Suite IDs Suspicious Circuits
Detection Window
Application Stage: Emulate w/Software INTs
Baseline Reference: "High-Level Spec."
Detection Window
Application Stage
Baseline Reference
Application Stage
Detection Window

Figure 4. Baseline reference, application stage, and vulnerability window for various MI detection methods.

VI. Recommendations for Future Work

We hope to scale up the demonstration to do a full suite of assertions on a more complex processor, and investigate the algorithmic complexity, specifically as it impacts area and power overhead added by the checkers. As the method matures, it may be useful for design tool manufacturers to incorporate a robust set of checker-generators for PSL.

VII. Acknowledgment

This research is funded in part by National Science Foundation Grant CNS-0910734.

References

[1] DARPA Trust in Integrated Circuits (TRUST), http://www.darpa.mil/mto/programs/trust/.

[2] Isabelle, http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[3] MIPS Technologies, http://www.mips.com/.

[4] OpenCores, http://opencores.org/.

[5] AMD Processors Undocumented Debugging Features and MSRs, http://www.woodmann.com/forum/archive/index.php/t-13891.html, December 2010.

[6] AMD Undocumented Machine-Specific Registers, http://cbid.softnology.biz/html/undocmsrs.html, December 2010.

[7] IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pages 1 –171, June 2010.

[8] Xradia Exhibits X-ray Vision for Semiconductor Failure Analysis, http://www.xradia.com/company/news/press-releases/2010-12-01.php, December 2010.

[9] Abarbanel, Beer, Gluhovsky, Keidar, and Wolfsthal. FoCs - Automatic Generation of Simulation Checkers from Formal Specifications. In *12th International Conference on Computer Aided Verification*, Chicago, IL, USA, July 2000. LNCS 1855, Springer.

[10] Abramovici and Bradley. Integrated Circuit Security: New Threats and Solutions. In *CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*, pages 1–3, New York, NY, USA, 2009. ACM.

[11] Agrawal, Baktir, Karakoyunlu, Rohatgi, and Sunar. Trojan Detection using IC Fingerprinting. In *Security and Privacy, 2007, IEEE Symposium on*, pages 296–310, Oakland, CA, USA, May 2007.

[12] Banga and Hsiao. Trusted RTL: Trojan detection methodology in pre-silicon designs. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*, Anaheim, CA, USA, June 2010.

[13] Boule and Zilic. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties. In *Eleventh Annual IEEE International High-Level Design Validation and Test Workshop, 2006*, Monterey, CA, USA, November 2006.

[14] Eisner. PSL For Runtime Verification: Theory and Practice. In *7th International Workshop on Runtime Verification*, pages 1–8, Vancouver, British Columbia, Canada, March 2007.

[15] Findenig. Behavioral Synthesis of PSL Assertions. Upper Austrian University of Applied Sciences. Master's thesis, 2007.

[16] Gastin and Oddoux. Fast LTL to Buchi Automata Translation. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 53–65, Paris, France, July 2001. Springer-Verlag.

[17] Hicks, Finnicum, King, Martin, and Smith. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. *Proceedings of the 31st IEEE Symposium on Security and Privacy. Oakland, CA, USA*, May 2010.

[18] Iman. *Step by Step Functional Verification with SystemVerilog and OVM*. Hansen Brown Publishing Company. San Francisco, CA, USA., 2010.

[19] Jin and Makris. Hardware Trojan Detection Using Path Delay Fingerprint. In *Hardware-Oriented Security and Trust, 2008. IEEE International Workshop on*, pages 51–57, May 2008.

[20] Jin, Kupp, and Makris. Experiences in Hardware Trojan Design and Implementation. In *Hardware-Oriented Security and Trust, 2009. IEEE International Workshop on*, pages 50–57, 2009.

[21] Johnson. Fake Chips Threaten Military. *San Jose Mercury News*, September 2010.

[22] King, Tucek, Cozzie, Grier, Jiang, and Zhou. Designing and Implementing Malicious Hardware. In *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–8. USENIX Association, 2008.

[23] Markoff. Old Trick Threatens Newest Weapons. *New York Times*, October 2009.

[24] Pnueli and Zaks. PSL Model Checking and Run-Time Verification Via Testers. In *Lecture Notes on Computer Science*, pages 573–586. Springer, 2006.

[25] Rad, Plusquellic, and Tehranipoor. Sensitivity Analysis to Hardware Trojans Using Power Supply Transient Signals. In *Hardware-Oriented Security and Trust, 2008. IEEE International Workshop on*, pages 3–7, Anaheim, CA, USA, June 2008.

[26] Schneider. Enforceable Security Policies. *ACM Transactions on Information System Security*, 3(1):30–50, 2000.

[27] Tiwari, Wassel, Mazloom, Mysore, Chong, and Sherwood. Complete Information Flow Tracking from the Gates Up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 109–120, New York, NY, USA, 2009. ACM.

[28] Wang, Tehranipoor, and Plusquellic. Detecting Malicious Inclusions in Secure Hardware: Challenges and Solutions. In *Hardware-Oriented Security and Trust, 2008, IEEE International Workshop on*, pages 15–19, 2008.